Computer Supported Cooperative Work (2009) 18:445–475
DOI 10.1007/s10606-009-9101-3

On The Roles of APIs in the Coordination of Collaborative Software Development

Cleidson R. B. de Souza[1] & David F. Redmiles[2]
[1]*Universidade Federal do Pará, Belem, Brazil (E-mail: cdesouza@ufpa.br);* [2]*University of California, Irvine, Irvine, CA, USA (E-mail: redmiles@ics.uci.edu)*

Abstract. The principle of information hiding has been very influential in software engineering since its inception in 1972. This principle prescribes that software modules hide implementation details from other modules in order to reduce their dependencies. This separation also decreases the dependency among software developers implementing these modules, thus simplifying the required coordination. A common instantiation of this principle widely used in the industry is in the form of application programming interfaces (APIs). While previous studies report on the general use and benefits of APIs, they have glossed over the detailed ways in which APIs facilitate the coordination of work. In order to unveil these mechanisms, we performed a qualitative study on how practitioners use APIs in their daily work. Using ethnographic data from two different software development teams, we identified three roles played by APIs in the coordination of software development projects. These roles are described using three metaphors: APIs as contracts, APIs as boundaries, and APIs as communication mechanisms. As contracts, APIs allow software developers to work in parallel and independently. As a communication mechanism, APIs facilitate communication among software developers by giving them something specific to talk about. At the same time, APIs establish the boundaries between developers, and, accordingly, what should be talked about. This paper also reports on problems the studied teams face when using APIs to coordinate their work. Based on these results, we draw theoretical implications for collaborative software engineering.

Key words: interfaces, application programming interfaces, coordination, collaborative software development, software engineering

1. Introduction

Software development is typical of a collaborative endeavor where several software engineers work together to achieve the same goal: the delivery of a software system on time, on budget and according to specification. As a cooperative effort, some of the problems faced by software developers are the same problems faced by professionals in other domains: communication breakdowns, coordination problems, lack of knowledge about colleagues' efforts, and so on (Schmidt and Simone 1996). In fact, researchers and practitioners have long recognized that breakdowns in communication and coordination efforts constitute a major problem in collaborative software development (Brooks 1974; Curtis et al. 1988).

To be able to successfully coordinate dozens or hundreds of engineers working in the construction of a complex, fundamentally invisible, and highly dependent

product is a challenging task. Several approaches have been proposed to facilitate this work, ranging from software development methods, processes, tools, principles, organizational strategies, and so on. Among these approaches, one of the most important and influential is the principle of *information hiding* (Parnas 1972). According to this principle, software modules should hide implementation details that are likely to change and expose only aspects that are less likely to change. The information-hiding principle is instantiated as several different mechanisms in programming languages including data encapsulation, separation of interface specifications and their implementation, and polymorphism (Larman 2001).

In this paper we are particularly concerned with interfaces and APIs. In Software Engineering an *interface* is the set of services provided by a software component, while a component's *implementation* describes the way these services are implemented (Ghezzi et al. 2003). A clear distinction between the interface of a component and its implementation is a key aspect of good software design. In general, several interface specifications are combined creating what is known as an Application Programming Interface (or simply API). For the purposes of this paper, we will adopt a definition of an API as proposed by des Rivieres (2004): "a well-defined interface that allows one software component to access programmatically another component and is normally supported by the constructs of programming languages." In a programming language like Java, an API corresponds to a set of public Java-interface specifications, methods, and classes. APIs are in widespread use in the industry (des Rivieres 2001; des Rivieres 2004).

Over the years, practitioners and researchers have recognized the benefit of interfaces and APIs to the coordination of software development work. For instance, Grinter and colleagues describe how:

> *"(...) interface specifications play the well-known role of helping to coordinate the work between developers of different components. If the designers of two components agree on the interface, then design of the internals of each component can go forward relatively independently. Designers of component A need not know much about the design decisions made about component B, so long as both sides honor their well-specified commitments about how the two will hook together." (Grinter et al. 1999)*

In fact, the relationship between interface specifications and the coordination of work has become such a well-known argument that can even be found in software engineering textbooks:

> *"If a design is composed of highly independent modules, it supports the requirements of large programs: independent modules form the basis of work assignments to individual team members. The more independent the modules are the more independently the team members can proceed in their work."(Ghezzi et al. 2003,p. 241)*

CSCW researchers have also recognized the importance of APIs in the coordination of development work. Grinter and colleagues (1999) for instance argue that interfaces are coordination mechanisms. A coordination mechanism is:

> *"a construct consisting of* a coordinative protocol *(an integrated set of procedures and conventions stipulating the articulation of interdependent distributed activities) on the one hand and on the other hand* an artifact *(a permanent symbolic construct) in which the protocol is objectified."* (Schmidt and Simone 1996) *[emphasis in the original]*

By using the two aspects of a coordination mechanism — artifact and protocol — as an analytical lens to look at studies of collaborative software engineering, we can conclude that previous research has essentially focused on the *artifact* aspect of APIs. For instance, there is increasing research in how to design APIs (Michi 2009), how to evaluate their usability automatically (de Souza and Bentolila 2009) or manually (Ellis et al. 2007), how to use APIs based on examples (Xie and Pei 2006), and so on. Some of this previous work describes limitations of APIs. For instance, Ellis and colleagues (Ellis et al. 2007) studied the usability of the Factory pattern (Gamma et al. 1995) and found out that the use of factories can and should be avoided in many cases. Another example is the work of Kizcales and colleagues (Kiczales 1996; Kiczales et al. 1997) who maintained that users of software components need to know some information about a component's implementation — and not just its API — in order to make appropriate decisions about whether to use the component and how to use it. For instance, consider a window system API that provides functionality of display, mouse-tracking, and so on. If an API user wanted to implement a spreadsheet using this API he would not be able to do so without major performance overheads because the window-system implementation was not tuned for this kind of use allocating excessive objects and memory (Kiczales 1996).

While very insightful and important, previous work on APIs and their limitations mostly ignores the *protocol* aspect of APIs. In other words, APIs certainly facilitate the coordination of software developers' work, and both practitioners and researchers certainly agree with that. However, naming an API a coordination mechanism actually *hides* the details of how the coordination takes place, i.e., the ways in which APIs help software developers to coordinate their work are taken for granted instead of being carefully inspected. *In this paper, we argue that the coordination facility provided by interfaces is to be studied, not assumed*. That is precisely our contribution: we unveil *the protocol* that goes along the APIs, i.e., the unwritten division of labor, rules, and conventions that an API implies. Instead of assuming that coordination is achieved using APIs in collaborative software development efforts, we illustrate how coordination through the usage of APIs is achieved by describing in details the ways in which software developers get their work done using APIs. By questioning assumptions about APIs, we were also able to observe, describe, and discuss problematic situations in the daily use of APIs.

The work presented in this paper is an extension to our previous work, where we document a number of roles, advantages, and disadvantages of APIs (de Souza et al. 2004a). That is to say, it is our intent with this current publication to further elucidate the common practice of using APIs to coordinate work, and in particular, the problems and the practices employed to compensate and manage the process. To support our arguments, we use ethnographic data from two software development teams, both coordinating their work using APIs but each having different goals, experiences, and management styles. In both teams, the implementation of the APIs is being developed in parallel with the code that uses these APIs[1]. Using grounded theory techniques (Strauss and Corbin 1998) to analyze the data, we identify three complementary roles for APIs. These roles were described using 3 metaphors: APIs as contracts, APIs as boundaries, and APIs as communication mechanisms. As contracts, APIs allow software developers to work in parallel and independently. As boundaries, APIs allow developers to focus on their work isolated from their colleagues. As communication mechanisms, APIs facilitate the communication among software developers by defining what developers should talk about. When simultaneously fulfilling these roles, APIs support the coordination of software developers' daily work allowing them to work in parallel, in isolation, and focused on specific parts of their colleagues' work. However, our results also present surprising side effects we observed, i.e., problems that one needs to pay attention to when using APIs. Arguably, the more important observation is that the adoption of APIs by the development teams studied created the illusion that developers could work independently of their colleagues without problems. Finally, our observations confirm a number of individual results in the literature but reveal particular details that provide a foundation for future research in collaboration and collaborative software engineering in particular. The foundation can support further exploration in software tools and management techniques.

The rest of the paper is organized as follows. The next section defines some key concepts surrounding APIs and briefly explains their adoption by industry. After that, Section 3 presents the teams studied, while Section 4 describes the research methods that we used in our study. Then, Section 5 describes the multiple roles played by APIs in the sites studied. Section 6 discusses the problems when dealing with APIs. After that, Section 7 presents a discussion and implications of our findings. Finally, we present our final conclusions.

2. Application Programming Interfaces

2.1. API definition

In order to further explain the concept of application programming interfaces, we need to first explain a couple of important software engineering principles. Separation of concerns, for example, is one the most important principles in software engineering

that allow one to deal with different individual aspects of a problem, so that it is possible to concentrate on each separately. When different parts of the same system are dealt with separately, a particular type of separation of concerns called *modularity* is used (Ghezzi et al. 2003). It is then necessary to define how to divide a software system in modules. Parnas (1972) proposed the information hiding principle that recommends how modules should be designed. According to this principle, software modules should hide implementation details that are likely to change and expose only aspects that are less likely to change, i.e., modules should be both "open (for extension, and adaptation) and closed (to avoid modifications that affect clients)" (Larman 2001). This principle is instantiated in programming languages as several different mechanisms such as data encapsulation, polymorphism, or interface specifications (also called APIs — application programming interfaces) (Larman 2001).

des Rivieres's API definition, presented in Section 1, describe APIs as interfaces between software components, among professional software engineers the term API is coming to mean any well-defined interface that defines the service that one component, module, or application provides to others software elements. In the rest of the text, we will use the terms component and module indistinctly, since they do not change the purpose of using APIs. Typically, in a programming language such as Java, an API corresponds to a set of public methods, classes and interfaces, and the associated documentation (in this case, javadoc files).

Finally, the word interface in the abbreviation is used to explicitly indicate that APIs are constructs that exist in the boundaries of different software components. These two (or more) components are often developed by different teams, and hardly ever individuals. An example of a well-known API is the Microsoft Windows API that allows a program to access and use resources of the underlying operating system such as file system, scheduling of processes, and so on. Of course in that usage, an API is being perceived in terms of its functional role, versus a social role in terms of coordination.

2.2. API adoption

APIs are largely adopted by industry because they support the separation of interface from implementation (Fowler 2002), i.e., they are a common way of hiding component specification and implementation details from users of those components (e.g. see (des Rivieres 2004)). The main advantage of this approach is the possibility of separating modules into public (the API itself) and private (the implementation of the API) parts so changes to the private part can be performed without impacting the public one. By using APIs, companies are able to provide functionality to thousands of developers (e.g., Java Swing API) in such a way that developers do not need to know how this functionality is implemented. As a consequence, these APIs can evolve without impacting developers.

In the rest of the text, we will adopt the terms API consumers and API producers. API consumers are software developers who write code with method calls to an API,

and API producers are software developers who write the API implementation. An API can be classified according to its consumers (Fowler 2002): when the consumers are programmers from different organizations — for instance, if the API provides services from a Web application — the API is called *published*, but if the consumers are internal to the company, or at least, known by API producers, the API is called *public*. In this paper, the APIs studied were public according to this classification, i.e., API producers and consumers were in the same organization, BSC. Furthermore, API providers were implementing the APIs while consumers were trying to use these APIs. This situation, while is not unique to BSC, is rather different from a more "traditional" view of APIs were an API is only made available to developers after it is fully implemented. Since performing these field studies and spending time analyzing data from the socio-technical perspective, we have noticed that other field sites had similar issues with APIs. Therefore, though this work is limited to two field sites, we believe the observations will be useful to any organization were APIs play a major role.

An important aspect of any API is stability. A stable API is not subject to frequent changes, therefore leveraging the promised independence between the API producers' and consumers' code. Changes in the API itself require changes in the API consumers' code because this code uses services provided by the API. This situation might become problematic if changes to the API happen too often. As a result, API consumers expect that APIs will not change often, and if it does happen, they also expect that these changes will not severely affect them. Recent work in software engineering tries to provide advice on how to properly change APIs so that the impact of those changes is minimized (Fowler 2002; des Rivieres 2004).

3. Research sites

Our fieldwork was conducted in a software development company that we will call BSC (a pseudonym). BSC is one of the largest software development companies in the United States with products ranging from operating systems to software development tools, including e-business and tailored applications. Two different teams were studied: MCW and MBL (also pseudonyms), each one of them is detailed below.

3.1. The MCW team

The first team studied, called MCW, is responsible for developing a client-server application that had not yet been released during the period of the study. The project staff includes 57 software engineers, user-interface designers, software architects, and managers, who are divided into five different sub-teams, each one developing a different part of the application. The sub-teams are designated as follows: lead, client, server, infrastructure, and test. The lead sub-team was comprised of the project lead, development manager, user interface designers, and so on. The client sub-team was developing the client side of the application, while the server sub-team was developing the server side of it. The infrastructure

sub-team was working in the shared components to be used by both the client and server sub-teams. Finally, the test sub-team was responsible for the quality assurance of the product, testing the software produced by the other sub-teams.

3.2. The MBL team

The second team we studied, MBL, was responsible for developing a mobile version of the same application developed by the MCW team. MBL developers wanted to use, as much as possible, the MCW APIs and implementation, but this was not always possible because of hardware constraints in the mobile device they were targeting.

The MBL project staff was divided into three major groups: user interface (UI) designers, software developers, and the quality assurance (QA) team. The staff was distributed over five different sites in three countries: North Carolina, US; Massachusetts, US; Beijing, China; Shanghai, China; and Taipei, Taiwan. To be more specific, user interface design and evaluation was performed by six professionals in North Carolina. The implementation was performed in all other sites, distributed as follows: nine developers in Massachusetts, five in Shanghai, five in Beijing, and four in Taipei. The quality assurance team was divided between the US and Chinese sites: three engineers were located in Massachusetts and six engineers in Beijing. The main coordination of the project and the project manager for this project were located in Massachusetts, where all the data were collected.

There was no intersection between members of the MCW and the MBL teams. Note that in the MCW team, software developers — those responsible for writing code — were split among different sub-teams, while in the MBL team all developers belonged to the same sub-team. We also interviewed three members of another team whose component provided services to the MBL application, but not to the MCW application.

3.3. The organizational approach to APIs

At the time of the study, BSC had recently adopted a strategy of developing reusable software applications. This strategy aimed to create applications (each one developed by a different project) from software components that could be used by other applications (projects) in the organization. In fact, both projects studied used several software components provided by other projects, which means that team members of both teams needed to interact with other software developers in other parts of the organization.

To facilitate the reuse program, BSC enforced the use of a reference architecture during the development of software applications. The BSC reference architecture prescribed the adoption of some particular design patterns (Gamma et al. 1995), but at the same time gave software architects across the organization flexibility in their designs. This architecture was based on tiers (or layers) so that components in one tier could request services only to the components in the tier immediately below them (Buschmann, Meunier et al. 1996). Data exchange between tiers was possible

through well-defined objects called “value objects.” Meanwhile, service requests between tiers were possible through Application Programming Interfaces (APIs) that hide the details of how those services were performed (e.g., either remotely or locally, with cached data or not, etc.).

In this organization, APIs were designed by software architects in a technical process that involves the definition of classes, method signatures, and other programming language concepts, and the associated documentation. Accordingly, each software component would have a public API through which its consumers could access the set of services provided by that component.

To be more descriptive, we asked how an API would look; one of the MBL developers defined it as follows:

> *MBL developer 04: Maybe for each component more — not more than 15 classes on average over the three components. ...*

Each software component and its respective API were developed by a different project team, and could be used by other projects teams in the organization. Most projects implemented different sets of services, therefore implementing several APIs. Despite their willingness to reuse software components, different teams in the company developed different software components that provided similar sets of services. For example, one team would provide access to a particular type of service implemented in one particular platform. Another team would also provide access to the same type of services in a different platform. In this case, these software components would provide similar APIs. To guarantee that APIs were consistent and that software components were indeed reused throughout the organization, each project team had a software architect responsible for the specification of the APIs. Weekly meetings of the organization’s software architects were used to monitor this work.

Despite the importance of the APIs in the BSC software development process, the organization had no established formal process to create, implement, deploy, and maintain APIs. In one of the meetings that we observed, developers from different groups discussed the lack of recommendations by the software architects on how to proceed when facing such issues. As a senior MCW developer pointed out: “All APIs need to look, feel, and smell the same.” This lack of an established process had already been identified by the software architects and was starting to be discussed in the software architects’ weekly meetings.

4. Research methods

In order to properly study the use of APIs by software developers, we adopted a qualitative research approach because this allowed us to focus on how individuals and groups experience, view and understand the world and, more importantly, construct meaning out of their experiences. To be more specific, we conducted an ethnographic study of the two software development teams, MCW and MBL, to understand how

APIs are used in practice. Field studies are one approach for qualitative research that requires researchers to spend time in the "field," the natural context where the phenomena take place. Only by doing that is it possible to obtain an adequate understanding of the phenomena being investigated (Fetterman 1997). Accordingly, the first author spent 11 weeks with the MCW team during the summer of 2003 conducting non-participant observation (Jorgensen 1989) and semi-structured interviews (McCracken 1988). We conducted 15 semi-structured interviews with members of all five sub-teams. The questions were designed to encourage the participants to talk about their everyday work, including work processes, collaboration and coordination efforts, problems, tools, and so on. Interviews lasted between 35 and 90 min. In addition, we collected different documents, including meeting invitations, product requests for software changes, and emails and instant messages exchanged among the software engineers. We were also granted access to shared discussion databases used by the software engineers. All this information was used in addition to field notes generated by the observations and interviews.

Before the analysis of the data, interviews were transcribed and field notes were typed. Both types of data, as well as the documents collected were integrated into a software tool for qualitative data analysis. All the material collected was analyzed using grounded theory techniques (Strauss and Corbin 1998). In other words, the three major steps proposed by the grounded theory — open coding, axial coding, and selective coding — were performed. The result was a theory grounded exclusively on the existing data that explains the roles of the APIs in the coordination of the work.

The grounded theory approach calls for an interplay between data gathering and analysis to develop an understanding of what is going on in the field. Basically, as the fieldwork progresses, hypotheses are generated, tested and modified according to the ongoing analysis of the data being collected. During our fieldwork, we eventually realized the fundamental role of APIs in the coordination of software developers' work. For instance, early in the data collection process a MCW software developer explained the work of the different sub-teams by describing their relationship with APIs:

> *"Our only work is to make these APIs work ... the client* [sub]-*team's* [work] *is to consume the APIs and create user interfaces."*

Accordingly, we collected more information about this aspect in order to get a broader understanding of the APIs' importance and to verify whether we had understood developers' work. Finally, the interviewees provided feedback on our interpretation of the roles of APIs. In fact, the importance of APIs in the coordination of the software developers was clearly recognized by members of the software development team, who agreed: "APIs are the heart of the whole exercise." This feedback was fundamental to improving our understanding of their work and to give us confidence in our results. As mentioned before, results of this initial work are reported in (de Souza et al. 2004b).

During the summer of 2004, the first author returned to BSC to study the MBL team. In this case, data were collected through document analysis and semi-structured interviews (McCracken 1988). Among other documentation, emails and instant messages exchanged among the software engineers were collected. We again were granted access to shared discussion databases used by the software engineers. This information was used in addition to notes generated by the interviews. We conducted 17 semi-structured interviews with members of all sub-teams from the different sites: some interviews were conducted face to face, and others were conducted by telephone, with one interview conducted by using instant messaging. We reused some of the questions used in the interviews with MCW team members, but we also explored communication, collaboration, and coordination efforts between their collocated and distributed colleagues, and between the MCW and MBL team members. Interviews lasted between 20 and 70 min.

Interviews were again transcribed and field notes were typed. All the data collected was integrated into the software tool for qualitative data analysis used before in order to reuse the same codes used in the analysis of the MCW data. In addition, during the analysis of the MBL data, new codes were created, while others were removed to clearly document the differences (or similarities) between the teams. This additional material was used to expand our understanding of the roles of APIs in the coordination of software developers' work. And again, MBL developers provided feedback on our interpretation of the roles of APIs and helped us to tease out aspects that were specific to the MBL team, in contrast to the MCW team. Results of our analysis are described in the following sections.

5. The roles of application programming interfaces

During our analysis of the data, patterns of general roles played by APIs began to emerge. Similarly to Smolander (2002) and other researchers in different areas, we use metaphors to describe these roles because they allows us to understand these roles "by means of other concepts that we understand in clearer terms" (Larkoff and Johnson 1980).

In our analysis, we identified three general metaphors for APIs in the software development processes of the MCW and MBL teams. These metaphors, or roles, are named as follows:

1. APIs as Contracts;
2. APIs as Boundaries; and
3. APIs as Communication mechanisms.

Some of these metaphors arose out of the simple analysis of the data. For instance, an informant suggested the first metaphor during an interview: "APIs are like a contract, they promise to deliver something." Other metaphors required a more careful analysis of the data. Each one of these metaphors is described in detail in the following sections.

5.1. APIs as contracts

As Parnas (1972) theorized, the information-hiding principle allows one module to hide its implementation details that are likely to change from its client modules. As discussed in Section 2, interface specifications, and APIs, are one implementation of this principle and, as predicted, allow different software developers to go about doing their work in parallel. Our analysis suggests that this parallelism is possible because, in practice, APIs can be seen as *contracts* between two parties, i.e., each developer can work independently because one party knows what will be delivered by the other party — it is specified in the contract. MCW and MBL software developers themselves defined APIs as contracts: according to MCW-developer-04 "*APIs are like a contract, they promise to deliver something*." As contracts, APIs minimize the coordination needs required to the successful construction of the software product.

Software developers' parallel work starts after the contract is established, i.e., there is a consensus about what the API is supposed to look like. In the MCW team, this happens after a formal design review meeting when all the interested parties discuss the API:

Researcher: Who is invited to those [API design review] *meetings?*

Informant: People on our [sub]-*team who are responsible for implementing the feature from the server side. The* [another project] *people who have implemented, whose APIs we have decided to share or use. Some of the management* [sub]-*team and the architects for our projects. They are the primary people who are involved in that meeting ...*

Researcher: What about QA [quality assurance]*?*

Informant: Yes, they are involved as well.

Researcher: Why did they go [to the API meeting] *if there is no implementation yet?*

Informant: Basically to give them an opportunity to participate in the process, to be aware of what is coming in the process so they can prepare for it, even though there was no implementation that day. If they had any questions or concerns or nothing else just to inform them that the service was coming on and they were ultimately involved with testing it. It gave them an opportunity to have a look early on, or it just came over the wall to them.

Usually the person scheduling the meeting was the API developer responsible for implementing the API, or his manager. The meeting is scheduled only when this developer had defined a technical strategy for implementing from scratch or reusing the API. This strategy would be preceded by several technical discussions (face-to-face, over email, IM, etc) between the developer and other software engineers and architects in the organization.

Design review meetings play an important role in such a large organization as BSC: in addition to allow the technical discussion about the API, they are also an opportunity for all software developers interested in a particular API to meet. This is particularly important because these developers still need to coordinate their work, despite the parallelism in their work allowed by APIs. For instance, members of the MCW test sub-team meet the developers who will implement the API (the API providers) during the meetings. Later, testers will email information to these API providers about how the APIs are going to be tested, with the intent of avoiding minor integration problems that could delay the development schedule.

In the MBL team, the definition of the API — and the establishment of the contract that goes along with it — is more informal because APIs, in this team, are located between software developers in the same team (see next section):

Researcher: So can he ask you for a new method, a new API — a new method in the API or something like that?

Informant: Yeah, at the beginning when I was doing the design, I was really talking with him and then I discussed what API he needs. So and then we come up with a set of API and then we start from there. But now it is very unlikely, he does not really ask for any more. He did not ask for a long, long time already.

After the API is defined, all interested parties are ready to work independently because they all expect that the established contract is going to be fulfilled. However, before that, one last step is necessary. As mentioned before, an API defines an interface, a set of services to be provided by a component. In order for an API to be usable, it is necessary to implement these services. Therefore, the MCW software architect defining the API provides a stub implementation of the API for those interested in the API. The purpose of this implementation is to allow API consumers to immediately start programming against this API. According to one MCW software architect:

"The first-pass delivery (...) is a shallow implementation, just enough to start some work but it does not really flesh out anything."

Software developers would refer to these stub implementations as "local APIs," in contrast to "remote APIs," which are the real APIs implemented by the server team[2]. Periodically, API providers replace parts of the "local APIs" by their real implementation often based on suggestions and needs of the API consumers: "(…) when it [the implementation] is ready, I replace the dummy code for the real implementation." More formally, MCW managers decide which deliveries each sub-team makes:

Informant: [A deliverable from the server sub-team] *is in form of milestones. We deliver certain pieces of the API. Actually, the way that the* [MCW] *project*

is meant to work was we were really expected to stub in the whole API so even if the functions did not do anything they were callable by the client [sub-team]. *Then, as the driver scheduled progresses various pieces of functionality are actually enabled in the server and the client can then actually use them.*

Researcher: So who defines which pieces are enabled?

Informant: Generally the team leads. The client [sub]-*team leads and the server* [sub]-*team leads which would be* [X] *and* [Y] *and the management* [sub]-*team. The management* [sub]-*team are the people who decide what we deliver for any given milestone … and when it should come in and when the schedules should be set to deliver certain pieces based on our inputs.*

A similar approach was adopted in the MBL team regarding the deliveries from the developer implementing the API to the developer who is consuming it:

MBL developer 03: [MBL developer 01] *does not — he did not wait for me to finish my implementation. So we kind of like agreed on interface, roughly to say, yes, this should work or might work. … And they can start to code his stuff. And by the time I'm done and he's done sort of like partly done and we can just — can start to use my implementation. But even before I provided my implementation, UI is already started working assuming this interface, this set of* [methods] *will work and he will be able to use — he would use this method …, which is how we did it.*

In short, APIs facilitate the coordination of the work only to the extent that they are seen as contracts, as promises that will eventually be fulfilled during the software development process. The coordination of the work is achieved by allowing software developers to work in parallel: APIs isolates the work that each [sub-]team or developer needs to perform. However, for that to happen, stub implementations of the APIs are necessary because they allow API consumers to have something to implement against.

In addition to allowing independent work, APIs also minimize the impact of changes in one module into other modules. Software developers are aware of that, but they are also aware that this is only possible to be achieved in their organization when APIs are used, because as contracts, they can be trusted. In fact, a developer in the MCW team asked another team in the organization to create APIs out of their internal interfaces. By doing that, he could make sure that he would not be affected by changes in this other team's code, because the team would have to guarantee the stability of the API:

"I formally made a request to them to expose the API and test the functionality that I needed". The [software component's name] *team had problems before because they used unexposed APIs. He learned that from the manager of the* [software component's name] *team when he was suggesting to not expose the API, but because of the "hard time" they had, he wanted to avoid it: "I don't want to be in this situation where there's a new version and I have to change* [my] *code".*

5.2. APIs as boundaries

As discussed in Section 3.3, in the BSC organization, each software application is created out of different software components, whereas each component provides different services to other components within the same application or in a different application. These services are made accessible through different APIs as specified in the reference architecture, i.e., APIs are the external boundaries of a software component.

We observed, during the first data collection period, that a unique software development team implements each software component. Furthermore, APIs were used to divide the work between sub-teams in the MCW project: for instance, there were six different APIs between the MCW server and the client sub-teams. In other words, APIs are not only boundaries between different parts of the software architecture, but also boundaries between the [sub]-teams: they define the limits of what is usually known about a [sub]-team and what needs to be done by each [sub]-team. For instance, being an API provider means to be a member of the sub-team who is implementing this API, and consequently to understand its implementation details. On the other hand, to be an API consumer means to be part of a different sub-team, one that does not need to know the API implementation details. In short, APIs are then reifications of the organizational boundaries: any two given teams and sub-teams that need to interact (i.e., that their code needs to interact with each other) in the BSC organization will do so through the appropriate set of APIs that will integrate their software components. As described by MCW-developer-13: "*APIs are the layers that should be between teams or the public…*"

Typically, complex components need to interact with several other components, meaning that several APIs will mediate the cooperation among members of these two [sub]-teams. As mentioned, MCW architects reported that there are, at least, six different APIs mediating the work between the MCW client and server sub-teams. In addition, each one of these sub-teams needed to interact with other teams in the organization because their components needed to somehow interact. As described by MCW-developer-12:

> *Researcher: ... The thing that struck me ... is that one way of doing that* [collaboration between teams of software developers] *is through APIs. Is there anything else? Am I missing the point?*
>
> *Informant: So you have to define what a team is. For example, in a first pass you have the client, infrastructure and server* [MCW] *teams. Another pass there is a* [MCW] *team and the* [other team name] *team. And even in another higher level there is now* [BSC product 1, which encompasses MCW and other teams], [BSC product 2], *and* [BSC product 3]. *Each of those teams has levels of collaboration and cooperation. The closer you are, the tighter it happens. ...*
>
> *Researcher: ... even according to those levels, the APIs seem to be the boundaries between the teams or not?*

Informant: Yes, I would say so.

Within the MCW team, we also observed that APIs were used to coordinate the work of software developers in the same sub-team. However, this was the exceptional case. During the second data collection period, we observed a different situation. In the MBL team, there were no sub-teams, except for one responsible for the synchronization between the mobile application and the desktop. As mentioned before, technical dependencies in the MBL team were also handled through the usage of APIs, but in this case, APIs were used to separate the work done by software developers *from the same team*. APIs were used to separate the work according to the reference software architecture: a developer A and a developer B would have an API between them, if A was working in a part of the software architecture that provided services to the part of the software architecture B was working on. Because MBL developers who are implementing and consuming the API — in the example above developer A and developer B, respectively — are part of the same sub-team, coordination among them was easy when compared to the MCW team. This is illustrated by the second quote in Section 5.2 and by the following example (more details in Section 6):

Researcher: ... I'm just wondering if you need something from [component 1] *on that or from* [component 2] *or from whatever and then that's not implemented yet ...*

Informant: We're at the stage now where enough of the building blocks are there where it's, "Okay, I need a method." You know just a scenario, "When am I going to be able to do this?" and he says two or 3 days, and I wait for him to deliver [i.e., to commit the code in the configuration management repository] *before I rely on the code.*

In short, in both teams, APIs divided the work of software developers according to the reference architecture; and, by doing that, they established boundaries between the developers implementing the different parts of the software architecture. However, in the MCW team APIs were also reifications of organizational boundaries so that developers interacting through APIs were from different teams.

5.3. APIs as communication mechanisms

The third role played by APIs in the coordination of software development projects was to facilitate the communication of software developers about their own work. For instance, the following quote resumes the division of labor between the client and server teams:

"Our only work is to make these APIs work ... the client team's [work] *is to consume the APIs and create user interfaces"* [member of the server team]

This description was explained by a senior developer to the author, but could be used to explain the division of labor to a novice programmer. Furthermore,

APIs allow different software developers to have specific pieces of code to talk about, creating a "common ground" for them. For instance, according to MBL-developer-03:

Researcher: And whenever you had to change [the API]*, so what would happen? He would ask for you to change something* [in the API]*? You would tell him you were changing something?*

Informant: Yes, if he starts an issue or problem, then he would [ping me] *— or sometime or just come to my office saying, "Well, I need this thing..." And we'll have a discussion and then say, "What groundwork ... Should be that way? from my point of view it should be done this way and that way."*

In addition to data from the interviews, we also collected extracts from email messages exchanged between developers where they discussed, for instance, whether one particular method had already been implemented, the exception raised by another method, and so on.

During their meetings, MCW developers also talked about their work by simply referring to the APIs they needed. For instance, in one of the several weekly meetings, we attended, we observed an advice given by the MCW client manager: "Please contact your server-side API provider". In a API design review meeting, one of the MCW developers asked: "have you gone over every single method to discuss what each one has to do?"

It is important to mention that communication during a software development project changes as the project progresses. At the time of the data collection of the MBL team, most of the code was already implemented:

Researcher: ... so because you are consuming those other pieces of code, so I'm just wondering if you need something from [component 1] *or from* [component 2] *and then that's not implemented yet. So, is there like — that happens a lot, or not?*

MBL developer 01: We're at the stage now where enough of the building blocks are there, where it's, "Okay, I need a method." You know just a scenario, "When am I going to be able to do this?" and he says two or 3 days, and I wait for him to deliver before I rely on the code.

5.4. Summary

We used three metaphors to explain the three roles — contracts, boundaries and communication mechanisms — that APIs play in the coordination of software development work. By fulfilling these different roles APIs provide important advantages to the coordination of software development work. As contracts, they allow software developers to work independently and minimize the impact that a developer has on his/her colleagues when he makes changes to his code. In addition, APIs facilitate the communication among software developers by giving

them something specific to talk about, and, at the same time, they establish the boundaries between developers, and, accordingly, what should be talked about. However, our data suggests that there are also limitations to the advantages provided by APIs and that their usage might be problematic both for technical and organizational/social reasons. All these aspects will be discussed in the following section.

6. Limitations of APIs

This section describes some problems that the MCW and the MBL project teams faced in the light of API adoption and usage.

6.1. Limits on parallel work

The adoption of APIs in the BSC organization created the illusion that developers could work in parallel without problems. As discussed before, this is a fairly common assumption among software researchers and practitioners. To be more specific, the adoption of "local" and "remote" APIs was the work-around used by MCW developers since the API was not available until later in the process, i.e., parallel work can happen when APIs are adopted if additional mechanisms are used[3]. This approach is limited, however; it only works in the early stages of development becoming problematic as work progresses:

> *"I think ... this* [the usage of stub implementations] *works to some extent. But as you push further along implementation dummy stuff starts not working. So, for example, the list displays stuff, just dummy stuff, that works, but as soon as you want to open one of those dummy stuff, there is no stuffy behind the dummy stuff so the list can not hand off to the launcher* [component] *that can not hand off to the* [component] *... you can not open up because there is really nothing that far…It is a matter of how deep does the dummy stuff goes. You dive a really bit and then, there is no more there. It kind of works in the start but as you go further along (...)"*

In order to handle this limitation, MCW client managers, software architects and software developers evaluate the "local" APIs during every weekly meeting. Their goal is basically to assess either if the "local" API can still be used (work can still proceed in parallel) or if it is time to use the "remote" API (parallel work has to stop and integration needs to start). Sometimes, API consumers can continue using the "local" APIs, which means that they will go on working without contacting their API providers. However, when the "remote" API is needed, the client manager will contact the server sub-team manager and suggest the API consumer contact his or her API provider. Note that there is an assumption here, which is that the API consumer knows who his or her API provider is. But this is not always true. Often, API providers are not aware of the

consumers of their APIs, and vice-versa. According to MCW-developer-07: "*In talking to them* [another BSC team] *directly, they don't even know that we have this deliverable by* [date]*.*" This was mentioned as an example of an API provider in the BSC organization not being aware of the existence of an API consumer in the MCW team. In addition, the following quote clearly illustrates the lack of knowledge from a MCW server developer about who the API consumers were in the MCW client sub-team:

> *"Researcher: Who depends on you?*
>
> *Informant: The client* [sub]*team. Our* [MCW] *client* [sub]*team. Whoever on that team is going to be writing that end of our software that we will need to* [perform some actions].
>
> *Researcher: Do you know who they are?*
>
> *Informant: I am not entirely sure. My guess would probably be either* [MCW client developer 01] *or* [MCW client developer 02]. *I know the whole client* [sub]*team and I think that they are sort of working on this area. At the stage that I am in now I am implementing everything on the server side and having been at the client* [sub]*team and telling them to turn this on."*

Another side-effect of adopting APIs was software developers' expectation that the task of integrating the API producer's and consumer's codes would proceed smoothly. According to a MCW manager: "*if we use* [N, a large number] *weeks for integration, then we're doing something wrong.*" However, in reality, problems arose during the integration period. In the MCW team, for example, several problems happened during the last organizationally scheduled integration period. This situation led both the client and server teams to adopt a "pre-integration" period before the official integration period. The manager of the server team also decided to assign a new hire to perform "smoke tests" to minimize integration problems.

The main reason why there are problems during the integration period, i.e., despite the amount of effort spent in the design of an API, is that an API is necessarily incomplete. That is, an API defines the syntactic aspects of an interface, however, it does not provide enough details about its implementation, and sometimes these details are necessary to the API consumer (Kiczales 1996; Kiczales et al. 1997) (see the discussion section). This is recognized by MCW developers:

> *"*[MCW-developer-04] *is complaining about a recent change in the API that he depends on. The new API was sent to him by email. He tells me that it is 'a stupid idea' because there is only the doc files but he can't understand the dataflow or how one class works with the others because there is no diagram or "source-code". After talking about the source code, he checks the configuration management tool to find out whether the source code of the new API is available, but it is not."*

In the quote above, it is clear that MCW-developer-04 is interested in additional information that is not provided with the API. For instance, the rationale for the API that was changed.

6.2. Problems of rigid boundaries

APIs divide the work necessary to develop software into two distinct parts: a *production* part responsible for implementing the API and a *consumption* part responsible for using this API. In the MCW team, the internal and external parts would belong to different teams or sub-teams, while in the MBL team the internal and external parts were part of the same team (see Section 5.2). In any case, APIs play the role of boundaries between software developers. In the MCW team, these boundaries were also organizational boundaries. As Mintzberg (Mintzberg 1979) discusses these organizational boundaries "work to the advantage of the organization, allowing each unit to give particular attention to its own special problems". However, exactly because of that, APIs in the MCW team caused several problems in the coordination of developers' work. For instance, we noticed that teams lacked awareness about other teams' work precisely because they belonged to different teams: they had different managers to report to, different meetings to attend to, sometimes even different schedules. Developers often did not even know who they were supposed to be aware of (de Souza and Redmiles 2007). A MCW server developer, when asked about who were his API consumers, replied: "I think I am supposed to provide an implementation to [pause]… But I am a littler unclear right now."

In the MCW team, this problem was remedied, to some extent, by the manager who maintained constant and intensive communication about their teams' progress and schedules. Additionally, an approach adopted by the MCW client and server teams was to pair developers (one from each team) according to the APIs they were working on: for each server team member responsible for implementing an API, there was a client team member who was the consumer for that API. This organizational solution failed in some occasions because API consumers did not want to appear to be pressuring their server developer counterparts. Similarly, we found out that in the server team, some software engineers were not aware of their client counterparts, i.e., those who would consume the API they were implementing. According to the software architect interviewed:

> "[You could observe] *in our team meeting yesterday and other ones… people seem to be reluctant to talk to their counterparts too much … in the sense that they feel they're bugging the other person … and that is a problem because, I mean, the reason why we are here … the reason we're getting paid, we are developing a product and that interaction* [between client and server developers] *needs to happen."*

One might think that this type of knowledge about their counterparts is not necessary during the initial stages of software development while team

members might still be able to work independently. However, this same software architect pointed out, this lack of awareness is problematic even in this occasion:

> *"People think there's somebody else doing something* [on the API] *and when, you know* [the API is needed] ... *it is an empty void because they did not step up and said: 'I tried to identify my server counterpart or my client counterpart or if there is anyone. We got a problem here!'"*

Note that API design review meetings play an important role in the coordination of the work of the MCW team because they are events that allow all software developers interested in a particular API to meet, potentially avoiding some of the problems described above. However, the design of the API, and consequently the API review meetings, occur well-before the implementation of the API starts. Changes in people's roles and assignments during the software development process therefore remove this knowledge about API consumers and producers.

In addition, by reifying organizational boundaries in the MCW team, APIs indirectly hindered the collaboration among members of different teams that were not paired. For instance, we observed that another team in the organization was responsible for implementing a component that provided services for both the MCW server and the infrastructure team. Members of these teams were not aware that they shared this dependency and were working in parallel in overlapping aspects of this task. One software engineer identified this issue and decided to talk to the members of the other team so that they all could align their efforts and avoid duplicate work.

In the MBL team, APIs were boundaries between software developers working in different parts of the software architecture, but within the same team. As members of the same team, there were no organizational barriers among software developers and, therefore, the problems described above were not observed nor reported by our informants. In fact, when asked, MBL-developer-10 described his communication with his API provider as follows:

> *Informant: ... and then I'll send her an email that, "Is it okay for me to — is that ready for me now?" Then she will tell me that, "Okay, it's ready," or "Sorry, it's not implemented yet."*

This quote is particularly relevant because MBL-developer-10 is located in Taipei and is a consumer of an API being implemented by another MBL developer located in Massachusetts suggesting that the geographical distribution did not hinder the collaboration between these developers.

Being in the same team also allowed MBL API consumers and providers to have access to the same code base, and therefore, API consumers could easily find out information about the status of the API implementation by just consulting the configuration management tool.

A different engineer, MBL-developer-13, described her communication about the API design:

> *Researcher: So can he* [the API consumer] *ask you for a new method, a new API — a new method in the API or something like that?*
>
> *MBL Developer 13: Yeah, at the beginning when I was doing the design, I was really talking with him and then I discussed what API he needs, and then we come up with a set of APIs and then we start from there.*

Furthermore, developers in the MBL team were aware of their colleagues so that API consumers and providers would talk to each other when necessary. For instance:

> *Researcher: So basically, in case this — assuming that she has to make a change in API, so is she — does she send you an email or something telling that she's changing those or…?*
>
> *Informant: Oh, no. She'll tell me. Well, because the problem is if she actually changed the API, she's going to break my code; my code won't compile, so…*
>
> *Researcher: she's going to send you an email or ping you or even stop by or something?*
>
> *Informant: Yeah, exactly. And I'd have to search through my code and — to make those changes and know that my code wasn't going to compile until it (inaudible) with hers.*

MBL-developer-04's quote below suggests that members of his team even pro-actively took action to avoid causing problems for their colleagues (in this case, avoiding to block their colleagues' work).

> *Researcher: So basically when you start implementing* [an API] *you have to choose which class you're going to do first, which services are you going to start implementing ... right?*
>
> *Informant: That's correct.*
>
> *Researcher: Do you negotiate with them* [the API consumers] *or do you tell them, so,'okay, we're only going to start by this part' or do they tell you 'you should start by this part' …?*
>
> *Informant: Well, typically, everybody wants their own changes; then, they don't care about other people's changes. So ... it's more — it's less about negotiation. Sometimes it's just that, okay, well, is the person that's requesting this change totally, totally blocked? And if they're blocked, that's got to be a high priority because someone is sitting there not having anything to do.*

6.3. The not-so-contractual nature of APIs

As discussed earlier, the stability of an API is important: since software developers see APIs as contracts, they do not expect these APIs to change. Therefore, whenever they change, this is a major cause of burden for them because that requires them to make changes in their code. This situation might be more or less problematic depending on the type and amount of changes in the API and the frequency in which these changes occur. Software developers of both MCW and MBL acknowledge that changes in APIs were inevitable. According to a MBL developer:

> *"… because we're dealing with such rapid development and things are going to evolve over time. So, it's expected that the interface is going to change. We didn't cast interfaces in concrete before we started code. We couldn't — we didn't have the luxury of being able to do that…"*

And according to a MCW developer:

> *"I've never seen a technical spec that describes functional requirements that have been implemented without changes."*

> *"while you're developing code, everything can change."*

Note that APIs did change despite all the discussion that took place during the API design (e.g., during the API review meetings). Furthermore, although software developers of both teams recognized the need to change APIs, only MCW developers reported this situation as problematic because of the frequency of changes. According to a MCW client developer:

> *"But what has happened is that the server team has been publishing capabilities and APIs on a milestone basis. Every 2 months we have a milestone. … which makes it more difficult to design against because there are some future stuff that we are working that we don't know quite how it is going to be delivered …"*

Changes in the MCW APIs were so frequent that client developers often would ask questions like: "Is the [name] API changing?" They would ask this question in their weekly meetings before starting to work in the API in order to avoid problems.

To minimize the problems caused by changes in the APIs, members of the MCW server team (the API producers), before changing an API, meet and negotiate these changes with members of the client team (the API consumers). This is only possible, however, when a MCW server developer is aware of his API consumers, a situation not so common (see Section 6.2). When the negotiation is not possible, API providers try to, at least, notify their clients about changes in the API. To complicate further, we observed that in some occasions client developers would be notified about changes in the API, but the

actual changes would not be delivered to them right away. As explained before, the client team needs the server APIs to be able to use them as "local" APIs, creating a temporary independence from the server's team. In some cases, the changes to the API are not spread in the organization. And since other teams also depend on the set of services that the component makes available through its API, this situation makes the design and implementation tasks much harder. As one software developer reported: "*this* [the task of designing using an evolving API] *is a total moving target*".

The instability of the APIs was not a problem for MBL developers. Reexamining a quote used earlier, API consumers reported:

> *Researcher: So basically, in case ... assuming that she has to make a change in the API, so is she — does she send you an email or something telling that she's changing those or…?*
>
> *Informant: Oh, no. She'll tell me. Well, because the problem is if she actually changed the API, she's going to break my code.*

6.4. Summary

Both MCW and MBL developers faced problems or limitations when using APIs to coordinate their work: parallel work between API consumers and providers was limited because developers did not know who their counter-parts were; by reifying organizational boundaries, APIs indirectly reduced communication between developers leading to lack of awareness about their colleagues' work; and the instability of some APIs caused coordination problems between developers. It is important to note that some of these problems arise out of interactions between technical constructs (APIs) and organizational aspects (team boundaries), clearly illustrating how software development is a socio-technical endeavor.

7. Discussion

7.1. Advantages provided by APIs

The use of interfaces or APIs in software development projects is a fairly common approach because of the technical (and social) advantages they provide. Our study explores and illustrates in depth some of these advantages:

- APIs minimize the impact one developer might cause into another;
- They allow software developers to work in parallel and independently; and
- Finally, they limit what one developer needs to know about the work that his colleagues are performing.

While these advantages are widely recognized by practitioners and researchers, little is known about *how* these advantages are achieved in software developers' daily work. Our data suggest that these advantages are

possible because APIs play three different roles: contracts, boundaries, and communication mechanisms. By simultaneously fulfilling these roles, APIs achieve the critical goal of facilitating the coordination of software development work. As contracts between parties, APIs allow parallel and independent work when accompanied by their "stub implementations". In addition, as any other contract, APIs are *negotiated* between the interested parties. This negotiation can be formal — during an API design review meeting as in the MCW team — or informal — through software developers' conversations as in the MBL team. MCW software architects bring the client and test teams into the API design review meeting to approve the API. This is similar to what has been observed by Grinter (Grinter 1999), who discusses how software architects need to convince other members of the organization to "buy in" to their designs. In the MBL team, discussion between API consumers and providers takes place in the course of their daily work. In any case, without this negotiation between the parties, parallel work will not be achieved, despite the use of APIs. Overall, in the BSC organization one can argue that designing an API is as much a social process as a technical process, involving communication, coordination, and negotiation (Bucciarelli 1996).

APIs can also be seen as boundaries between developers, i.e., they limit what one developer needs to know about the work that his colleagues are performing. In the MCW team, these technical boundaries were also aligned with organizational boundaries establishing a very clear distinction about who is providing and who is consuming an API. As boundaries, APIs "work to the advantage of the organization, allowing each unit to give particular attention to its own special problems" (Mintzberg 1979). They help developers to work independently, to go about doing their work focusing only on the required information from their colleagues. In fact, APIs can also be seen as communication mechanisms that allow software developers to communicate more effectively about their work, since they provide a focus on *what* specifically should be talked about. By pairing developers from different teams, MCW managers also reduced the communication and coordination needs regarding API use: one client developer needs to engage in communication with one server developer only.

It is important to mention that the need for independent work is a common theme in software engineering research and practice. Independent work is necessary because of the several interdependencies that occur in these efforts: between tasks, between different artifacts, and within different parts of the same artifact. Several different approaches have been proposed including the principle of information hiding (Parnas 1972), and design by contract (Meyer 1992) just to mention a few. However, as illustrated by the study reported here and other studies (McDonald and Ackerman 1998; de Souza et al. 2003; Grinter 2003; Sarma et al. 2003), this isolation might hinder communication and coordination. This is discussed in the following section.

7.2. Problems when using APIs

Our data also illustrate some of the "side effects" that arise out of the adoption of APIs by a large organization. As presented in Section 6, parallel work is possible but limited. In fact, MCW software developers need to constantly question whether this parallel work should still take place. The integration period — the period in which the parallel work carried by two software developers must be brought together — was particularly problematic for the MCW team in the previous project iteration and took longer than expected. This problem led the server sub-team manager to allocate human resources to work to facilitate the next integration period. As one can expect, integration problems also happened when APIs changed and those changes were not broadcast to all interested parties. This situation was fairly common in the MCW team, since in this case, APIs reified organizational boundaries. This allowed software developers to work independently, but, at the same time, led these developers to "ignore" their colleagues working with the API counterparts hindering the collaboration among these developers. In short, MCW developers working with one part of the API (either as an API provider or consumer) were not aware of their colleagues working with the other part of the API. Therefore, when APIs changed, information about these changes were not properly propagated and caused coordination problems. Furthermore, in the MCW team integration problems occurred even when APIs did not change often because developers were still not aware of their API counterparts.

Grinter (1998) names the process of integrating parallel work *recomposition*, i.e., the work of putting all pieces together to create a software artifact. This work is required in any software development effort because the initial process of decomposition of a software system creates social relationships among the stakeholders that need to be maintained during the whole development process; otherwise the software cannot be later recomposed [*ibid.*]. In other words, our results illustrate that when APIs reify organizational boundaries, recomposition is harder to be achieved because it requires crossing organizational boundaries (Thompson 1967). They also show a technical solution (APIs) interacts with organizational aspects creating barriers for coordination.

Meanwhile, in the MBL team, APIs reified boundaries between developers who belonged to the same team, which did not lead to problems in the information flow. In fact, we have no data reporting coordination problems because of changes in the APIs. On the contrary, the data that we presented suggests that these MBL developers were able to successfully handle API changes. That was true even when the API producer and consumer were located in different geographical locations. A possible explanation for that is that the dependency between API producers and consumers forced these developers to coordinate their work, despite the geographical distance between them. This result contrasts with Grinter's work (Grinter et al. 1999) that describe the usage of interface specifications as a mechanism to coordinate a distributed software development project in the Delta organization. In their study, one of the problems faced by Delta was that components evolved independently making it hard to align

features during the integration period. In other words, Grinter describes problems faced by a distributed project because of changes in the interfaces, while our data suggest that, even when interfaces change in distributed projects, this might not be problematic as long as both parties are part of the same team and, therefore, are able to coordinate their work effectively (as in the MBL case).

Note that in both the MCW and MBL teams, APIs were used to isolate the same software components as specified in the BSC reference architecture. That is to say that the technical dependencies were very similar since these teams were also developing the same application for two different domains (desktop vs. mobile). Our data suggest that the coordination problems in the MCW team result as much from the organizational boundaries between team members as from dependencies between the software components. Similarly, despite the geographical distance, MBL developers were able to successfully coordinate their work because they had no organizational boundaries.

The discussion presented in Sections 7.1 and 7.2 suggests that APIs do facilitate the coordination of software development work (as theorized by Parnas (Parnas 1972) and assumed by other researchers (Herbsleb and Grinter 1999; Ghezzi et al. 2003). However, work-arounds are necessary (e.g., using stub implementations, pairing developers) for that coordination to be effective. Furthermore, it is necessary to make sure that software developers implementing and consuming the APIs engage enough in communication and coordination to make sure that both parts of the API — providers and consumers — are properly aligned, which will eventually lead to smooth work integration, or easy recomposition. This need for communication between software developers implementing dependent parts of the software is discussed in the following section.

7.3. Software structures and coordination

Over 40 years ago Conway (1968) had already recognized that the structure of the system mirrors the structure of the organization that designed it, a relation known as Conway's Law. According to him, in any design process, several design options are not made available to an organization because they do not reflect communication patterns of its members. Conway argues that the system structure will be overridden by the communication structure of the organization because the communication needs of those doing the work are inevitably reflected in the system. Conway's argument should not be understood as a prediction, because organizations are not immutable, they might change to facilitate the coordination of product development (Thompson 1967). Instead, Conway's proposal should be interpreted as advisory, an organizational pattern (Coplien and Harrison 2005, pg. 192), suggesting that software development can be facilitated by aligning the organizational structure and the software architecture.

As mentioned before, 4 years later, Parnas proposed the principle of information hiding (Parnas 1972), which minimizes communication needs among software

developers by restricting the information they exchange. Parnas suggested that by reducing dependencies between modules, it is possible to reduce software developers' dependencies on one another, creating a managerial advantage (Parnas, 1972). Taken together, Parnas' and Conway's arguments suggest that software dependencies *shape* the coordination and communication activities performed by software developers, and, at the same time, these dependencies *reflect* these coordination and communication activities (de Souza and Redmiles 2009).

The results of this paper corroborate this observation: API providers and consumers need to coordinate their efforts, even though they are using APIs that reduce communication needs between software developers. In the BSC organization, APIs were the points of interaction between API consumers and providers, i.e., the code from these developers interacted through the API. According to Parnas and Conway, these points should be accompanied by communication and coordination activities. In the MCW team, this did not happen because of organizational boundaries and even though API consumers and providers were collocated in the same city. On the other hand, in the MBL team this communication and coordination took place even though developers where located in distributed countries.

What is more interesting, however, is the temporal aspect of this communication and coordination needs. In the beginning of the project, strong communication and coordination are required to negotiate the APIs (through formal meetings or informal communication). Then, communication is reduced, or becomes less relevant, while API consumers and producers go about doing their work independently, as we discussed earlier with the help of stub implementations and other work-arounds. After that, communication and coordination become relevant again to guarantee the smooth integration of the work, the recomposition of the software system. In sum, it is possible to observe a "fluctuation" of software developers' communication and coordination needs as the project progress. This result is particularly relevant in the light of current approaches aiming to match technical dependencies needs and communication and coordination needs among software developers. Examples of this approach include software tools and conceptual approaches. An example of this approach is the Ariadne tool (Trainer et al. 2005; de Souza et al. 2007) that aims to identify dependencies between software developers from dependencies in the code they are developing. A conceptual approach is the idea of socio-technical congruence (Cataldo et al. 2006), which aims to explore the match between coordination requirements and actual coordination. A recent focus of these approaches is the identification of the "right" set of technical dependencies that drive the coordination work of software developers (Cataldo et al. 2008). Furthermore, these approaches assume that the technical dependencies are somewhat stable, i.e., they do not change over time. Our results show that it is necessary to identify not only the "right" set of the technical dependencies, but also *when* these dependencies are relevant. For instance, a technical dependency between an API client and the API implementation is not be relevant, at least for a while, after the API is negotiated between the parties. Another result from our work is that these "right" dependencies can be easily defined

by identifying the APIs that exist among the different software components. This approach is more efficient than syntactic dependencies (like call-graphs), which require analysis of the entire source-code, or evolutionary dependencies (Gall et al. 1998), which require analysis of the change history of the project.

The concept of awareness has come to play a central role in CSCW research (Schmidt 2002). In fact, this concept has led to different venues of research, from computational tool support, such as media spaces and event propagation mechanisms (Fitzpatrick et al. 2002), to ethnographic studies of work (Heath et al. 2002). Some of these studies discuss the role of awareness in settings as varied as ship bridges (Hutchins 1995a), aircraft cockpits (Hutchins 1995b), and transportation control rooms (Heath and Luff 1992). In particular, recent work has shown the importance of awareness in software development as well (see (Grinter 1995), (Grinter 1998), (Teasley et al. 2000),(de Souza et al. 2003)). Based on this empirical evidence, tools have been built in the last few years to support this approach (such as Jazz (Cheng et al. 2003a; Cheng et al. 2003b) and Palantír (Sarma et al. 2003)). This study builds upon this previous work by providing information about *what* information software developers need to be aware of. That is, API consumers need to be aware of changes in the API that they are using because the code that they are writing depends on it. Furthermore, both API consumers and providers need to be able to clearly identify each other, and be aware of each other because their work needs to be aligned. This need for information about their colleagues crosses organizational barriers, since an API might be written by a developer from a different software development team. This is in contrast to what Grinter (1998) suggests in her discussion about organizational awareness. She suggests that team information should be aggregated, while we present evidence that information about a single developer from another team is also necessary.

8. Conclusions

This paper investigates APIs as socio-technical constructs that facilitate the coordination of software development projects. APIs support coordination in software engineering by acting as contracts, boundaries, and communicational devices. By examining these roles, we identified strategies required for the successful coordination of the project. We also uncovered coordination problems that needed to be properly handled so that APIs' advantages could be fulfilled. In particular, our data show that when APIs reify organizational boundaries, communication regarding APIs is less effective.

APIs can be used to explain, enable and enforce the division of labor in the BSC organization. In fact, MCW and MBL informants *explained* their work in terms of API providers and consumers. APIs *enabled* providers and consumers to go about doing their work independently and in parallel, since these same APIs *enforced* boundaries between these software developers. Our results also illustrate how APIs create the fiction that developers can work completely independently of their

colleagues without problems. But we also found that with appropriate communication, APIs can support the coordination of large software development efforts.

Acknowledgments

The first author was supported by the Brazilian Government under grants CAPES BEX 1312/99-5, CNPq 479206/2006-6, CNPq 473220/2008-3, and by the Fundação de Amparo à Pesquisa do Estado do Pará (FAPESPA) through "Edital Universal N.° 003/2008". Both authors received support for this work from the U.S. National Science Foundation under grants 0534775 and 0205724, and by an IBM Eclipse Technology Exchange grant. Both authors wish to thank MCW and MBL teams for their participation, the reviewers for their detailed and careful comments, and Matt Bietz for several careful proof readings and critiques.

Notes

1. This might not be true for other software development teams.
2. These APIs are called "remote" because when the application is released, they will be located in a remote machine, an application server. Note that "local" and "remote" APIs are in fact the same APIs; the unique distinction between them is their *implementation*.
3. As discussed in Section 5.1, "local APIs" are the dummy implementations that were provided with the APIs, and the "remote APIs," were the real APIs implemented by the server team.

References

Brooks, F. P. (1974). The mythical man-month: Essays on software engineering, Addison-Wesley.

Bucciarelli, L. L. (1996). *Designing engineers*. Cambridge: MIT.

Buschmann, F., R. Meunier, et al. (1996). *Pattern-oriented software architecture: A system of patterns*. Chichester, West Sussex, UK, Wiley.

Cataldo, M., J. D. Herbsleb, et al. (2008). Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. Kaiserslautern, Germany, ACM.

Cataldo, M., P. A. Wagstrom, et al. (2006). Identification of coordination requirements: implications for the design of collaboration and awareness tools. *20th Conference on Computer Supported Cooperative Work*. Banff, Alberta, Canada, ACM Press.

Cheng, L.-T., De Souza, C. R. B., et al. (2003a). Building collaboration into IDEs. Edit ->Compile ->Run ->Debug ->Collaborate? *ACM Queue, 1*, 40–50.

Cheng, L.-T., S. Hupfer, et al. (2003b). Jazz: a collaborative application development environment. *ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, Anaheim, CA, USA, ACM Press.

Conway, M. E. (1968). How do committees invent? *Datamation, 14*(4), 28–31.

Coplien, J. O., & Harrison, N. B. (2005). *Organizational patterns of agile software development*. Upper Sadle River: Pearson Prentice Hall.

Curtis, B., Krasner, H., et al. (1988). A field study of the software design process for large systems. *Communications of the ACM, 31*(11), 1268–1287.

de Souza, C. R. B. and D. Bentolila (2009). Automatic evaluation of API usability using complexity metrics and visualizations (to appear). *New Ideas and Emerging Results — International Conference on Software Engineering*. Vancouver, B.C., Canadá, IEEE Press.

de Souza, C. R. B. and D. Redmiles (2007). The awareness network: Should I display my actions to whom? And, whose actions should I monitor? *European Conference on Computer-Supported Cooperative Work*. Limerick, Ireland, Springer: 99–117.

de Souza, C. R. B. and D. Redmiles (2009). On the alignment of organizational and software structure. *Handbook of Research on Socio-Technical Design and Social Networking Systems*. B. Whitworth and A. de Moor, IGI Publications. 1: 93–103.

de Souza, C. R. B., S. Quirk, et al. (2007). Supporting collaborative software development through the visualization of socio-technical dependencies. *ACM Conference on Supporting Group Work*, Sanibel Island, FL, ACM Press.

de Souza, C. R. B., D. F. Redmiles, et al. (2003). "Breaking the Code", Moving between private and public work in collaborative software development. International *Conference on Supporting Group Work (GROUP'2003)*, Sanibel Island, Florida, USA.

de Souza, C. R. B., D. Redmiles, et al. (2004a). How a good software practice thwarts collaboration-The multiple roles of APIs in software development. Foundations of Software Engineering, Newport Beach, CA, USA, ACM Press.

de Souza, C. R. B., D. Redmiles, et al. (2004b). Sometimes you need to see through walls— a field study of application programming interfaces. *Conference on Computer-Supported Cooperative Work*, Chicago, IL, USA, ACM Press.

des Rivieres, J. (2001, May 18, 2001). "How to Use the Eclipse API." Retrieved March 9., 2004, from http://www.eclipse.org/articles/Article-API%20use/eclipse-api-usage-rules.html.

des Rivieres, J. (2004). "Eclipse APIs: Lines in the sand." EclipseCon Retrieved March 18, 2004, from http://eclipsecon.org.

Ellis, B., J. Stylos, et al. (2007). The factory pattern in API design: A usability evaluation. *Proceedings of the 29th international conference on Software Engineering*, IEEE Computer Society: 302–312.

Fetterman, D. M. (1997). *Ethnography step by step*. Thousand Oaks: Sage Publications.

Fitzpatrick, G., Kaplan, S., et al. (2002). Supporting public availability and accessibility with Elvin: Experiences and reflections. *Journal of Computer Supported Cooperative Work, 11*(3–4), 299–316.

Fowler, M. (2002). Public versus published interfaces. *IEEE Software, 19*(2), 18–19.

Gall, H., K. Hajek, et al. (1998). Detection of logical coupling based on product release history. *Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society.

Gamma, E., Helm, R., et al. (1995). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley: Reading.

Ghezzi, C., M. Jazayeri, et al. (2003). Fundamentals of software engineering, Prentice Hall.

Grinter, R. E. (1995). Using a configuration management tool to coordinate software development. *Conference on Organizational Computing Systems*, Milpitas, CA.

Grinter, R. E. (1998). Recomposition: Putting it all back together again. *Conference on Computer Supported Cooperative Work (CSCW'98)*, Seattle, WA, USA.

Grinter, R. E. (1999). *System architecture: Product designing and social engineering. work activities coordination and collaboration*. ACM: San Francisco.

Grinter, R. E. (2003). Recomposition: Coordinating a web of software dependencies. *Journal of Computer Supported Cooperative Work, 12*(3), 297–327.

Grinter, R., J. Herbsleb, et al. (1999). The geography of coordination: Dealing with distance in R&D work. *ACM Conference on Supporting Group Work (GROUP '99)*, Phoenix, AZ, ACM Press.
Heath, C., & Luff, P. (1992). Collaboration and control: Crisis management and multimedia technology in London underground control rooms. *Journal of Computer Supported Cooperative Work, 1*(1–2), 69–94.
Heath, C., Svensson, M. S., et al. (2002). Configuring awareness. *Journal of Computer Supported Cooperative Work, 11*(3–4), 317–347.
Herbsleb, J. D. and R. E. Grinter (1999). "Architectures, coordination, and distance: Conway's law and beyond." IEEE Software: 63–70.
Hutchins, E. (1995a). *Cognition in the wild*. Cambridge: MIT.
Hutchins, E. (1995b). How a cockpit remembers its speeds. *Cognitive Science, 19*, 265–288.
Jorgensen, D. L. (1989). *Participant observation: A methodology for human studies*. Thousand Oaks: SAGE.
Kiczales, G. (1996). Beyond the Black Box: Open implementation. *IEEE Software, 13*(1), 8–11.
Kiczales, G., J. Lamping, et al. (1997). Open implementation design guidelines. *International Conference on Software Engineering*, Boston, MA, USA, IEEE Press.
Larkoff, G., & Johnson, M. (1980). *Metaphors we live by*. Chicago: The University of Chicago.
Larman, G. (2001). Protected variation: The importance of being closed. *IEEE Software, 18*(3), 89–91.
McCracken, G. (1988). *The long interview*. Thousand Oaks: SAGE.
McDonald, D. W. and M. S. Ackerman (1998). Just talk to me: a field study of expertise location. *Conference on Computer Supported Cooperative Work (CSCW '98)*, Seattle, Washington.
Meyer, B. (1992). Applying "Design by Contract". *IEEE Software, 25*(10), 40–51.
Michi, H. (2009). API design matters. *Commun. ACM, 52*(5), 46–56.
Mintzberg, H. (1979). *The structuring of organizations: A synthesis of the research*. Englewood Cliffs: Prentice-Hall.
Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM, 15*(12), 1053–1058.
Sarma, A., Z. Noroozi, et al. (2003). Palantír: Raising awareness among configuration management workspaces. *Twenty-fifth International Conference on Software Engineering*, Portland, Oregon.
Schmidt, K. (2002). The problem with 'Awareness' — introductory remarks on 'awareness in CSCW'. *Journal of Computer Supported Cooperative Work, 11*(3–4), 285–298.
Schmidt, K., & Simone, C. (1996). Coordination mechanisms: Towards a conceptual foundation of CSCW systems design. *Journal of Computer Supported Cooperative Work, 5*(2–3), 155–200.
Smolander, K. (2002). Four metaphors of architecture in software organizations: finding out the meaning of architecture in practice. *In Proceedings of the First International Symposium in Empirical Software Engineering*, Nara, Japan, IEEE Press.
Strauss, A., & Corbin, J. (1998). *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Thousand Oaks: SAGE.
Teasley, S., L. Covi, et al. (2000). How does radical collocation help a team succeed? *Conference on Computer Supported Cooperative Work*, Philadelphia, PA, USA, ACM Press.
Thompson, J. D. (1967). *Organizations in action: Social sciences of administrative theory*. New Brunswick: Transaction Publishers.
Trainer, E., Quirk, S., et al. (2005). *Bridging the gap between technical and social dependencies with Ariadne*. San Diego: Eclipse Technology Exchange.
Xie, T. and J. Pei (2006). MAPO: Mining API usages from open source repositories. *International Workshop on Mining Software Repositories*, Shanghai, China.